Component types and communication channels recovery from Java source code

Pascal André, Nicolas Anquetil, Gilles Ardourel, Jean-Claude Royer
LINA - EMN / University of Nantes, France
Email:(pascal.andre,gilles.ardourel)@univ-nantes.fr

Petr Hnětynka, Tomáš Poch
DSRG - Charles University, Prague, Czech Republic
Email: (hnetynka,poch)@dsrg.mff.cuni.cz

Dragoş Petraşcu and Vladiela Petraşcu
LCI - Babeş-Bolyai University, Cluj-Napoca, Romania
Email: (vladi,petrascu)@cs.ubbcluj.ro

Abstract

Software architecture erosion is a general problem in legacy software. Because they don't know or don't understand the original architectural intent, maintainers introduce changes that violate the intended architecture and properties. To fight this trend, component models and languages are designed to try to make explicit, and automatically enforceable, the architectural decisions in terms of components, interfaces, and allowed communication channels between component interfaces. But, what about existing systems written in traditional (e.g. object-oriented) languages? To help maintainers work on such systems, we explore the possibility of extracting architectural elements (components, communications, services, ...) from the source code. Some extraction heuristics are proposed and experimented on several implementations of a non-trivial system.

1 Introduction

Architectural erosion is the process by which a system's architecture gradually degrade as maintainers make changes to the system that violates the original architectural intents. This happens because maintainers are not aware of these intents or do not completely understand the system. The result is that the system becomes gradually more difficult to maintain as communication channels (*e.g.* method calls) are established among all parts of the system. It becomes difficult to make any significant change without having to change many parts, apparently unrelated to the issue.

To fight this trend, new languages and development methods are proposed that make explicit some architectural decisions in the source code (for the benefit of the programmers) and allow automatic verification and enforcement of these decisions, either at compile or execution time. For example, initiatives like ArchJava [**?**] extend the Java language with architectural *component types*, in and out *ports* on the components, that allow to establish explicit *connections* between the components, ... One interesting property is to be able to statically check communication integrity [**?**] that is to say, ensure that implemented components do not communicate between themselves in ways that would violate the intended control flow rules of the architecture.

In [**?**], Abi-Antoun *et al.* report their experience in manually re-engineering a conventional Java legacy system into a system with explicit definition of architectural components and allowed communication between them. Other worthwhile, but less ambitious, objectives could be: to help matching a concrete implementation to an abstract specification; or to help prolongating the life of the architecture by making explicit the components it contains, the communication between them, the provided and required services, etc.

As always in reverse engineering, one cannot expect a "random" application to follow strict development patterns, for example with clear separation of communications, data types, components types, etc. In this paper we assume an intermediary position, where we suppose that an application was developed with componentization in mind, but not necessarily with the required rigor. This would be the case for example, when one designs an architectural model, with proper specification of components and allowed communications, but implement the application with typical industrial approaches such as CCM, EJB, or OSGI that focus on

the runtime infrastructure, but provide little support for automatic verification of properties.

We explore the possibilities of automatic reverse-engineering of such an application toward a more formal model by extracting the component types it contains and making explicit the communication channels between them.

The paper is organized as follows. Section 2 gives the context of the project and a more formal definition of our goals. We then present our method to extract components from source code (Section 3). In Section 4, we present experimentation of our tool on Java applications and its results. We present the related work in Section 5 before concluding the paper and discussing future work.

2 Project

2.1 CBSE

Component Based Software Engineering (CBSE) tries to improve software development practice by proposing a development model where systems are assembled from components rather than programmed from scratch. CBSE claims to reduce development costs and improve the reliability of the resulting system. We will propose here a basic and generic definition of the main CBSE elements. There are many proposed models that implement these elements in specific ways, and add others (*e.g.* the notion of ports). We will not enter into these specificities.

In CBSE, systems are built from reusable *software components*, offering or requiring *services*, and with well defined *communication channels* between them. Bosch, Szyperski and Weck [**?**] propose the following definition of components: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." This implies that:

- A software component is a unit, that is to say it is cannot be divided and it is self contained (except for its declared required interface);
- it specifies an interface (or interfaces) of services it can provide, and it is bound by contract to implement at least this (these) interface(s);
- it specifies context dependencies, that is to say services it requires to work properly, its context dependencies are limited to, at most, this specification.
- it may be part of a larger *composite component*.

The specification of the provided and required services make up the *interface* of the component. Typically, a *service* will be implemented by a routine or a method accessible from outside the component. These properties make any component substitutable by other components that provide the same services.

Note that this definition of component differs from the usual definition used in reverse engineering and component recovery. Koschke [**?**], for example, defines components as one of: *Abstract Data Object*, a group of global variables and constants together with the routines that access them; *Abstract data type*, an abstraction of a data structure (a user-defined type) and all the type's valid operations on that data structure; *Hybrid Component*, an abstract data type that uses global variables to save state information, *Set of related routines*, a set of routines that together perform a logical functions, that is, have functional cohesion.

In comparison with the former definition, this one do acknowledge the fact that they are units, but it only focuses on the structure of the components in terms of atomic data (variable) and atomic services (routines), whereas the former definition allows to compose composite components from existing ones. However, the services provided are not explicitly specified, and there is no mention of the services required. This problem was also identified by Chouambe *et al.* [**?**]. In this paper we will use the first definition of components.

Concomitant to CBSE is the notion of component based architecture. A component based architecture formerly defines how components will be composed to form the system. It includes the specification of the components and their interface (provided and required services), and the *communication channels* between them, that is to say from what other components a component may require services (and which one), and to what other components it may provide services (and which one). Two components can communicate only if a communication channel has been formerly defined between them. This is one of the strength of the approach that allows to explicitly specify and automatically check some of the architectural decisions, thus actively limiting the chances of architecture drift. Allowed communications depend on the structure of the components: communications are allowed between a component and its parent (composite) component or between two siblings (subcomponents of the same composite component).

2.2 Project Overview

If component architecture models and languages allow to specify communication between components and check it statistically, they seem to have yet a limited impact on the practice. On the contrary, industrial approaches such as CCM, EJB, or OSGI focus on the implementation, they have strong and mature runtime infrastructure, but provide little support for automatic verification of the correctness of components' usage. As a result, applications often define flat component structure that do not allow checking the in-

tegrity of communication for example. Applications may be formally specified with an "academic" model but then implemented with the industrial approach. This leads to a mismatch between component specification and component implementation, where changes may happen on one level that would not be made on the other. This gap is called architecture erosion [**?**].

Our research takes place within the broader context of the Econet international project[1] which aims at establishing a link between component implementation — that could be called the *concrete* model — and component specifications — that could be called the *abstract* model. The concrete model can be any object-oriented application but we currently work with Java applications. There are a number of possible abstract models, in this paper we will assume a generic position with as little commitment as possible to any particular proposition.

This research contributes to a more general framework that the Econet project tries to establish:

- A common component meta-model that addresses both the problem of handling several specific component models (*e.g.* SOFA [**?**], Kmelia [**?**], KADL [**?**], FRACTAL [**?**]) in a generic way and the problem of linking abstract models and concrete code. The meta-model also provides the data structure to store the traceability links between models and code, and a set of rules to check the abstract models well-formedness.

- The structure abstraction tool (the research reported in this paper) extracts and infers architectural and typing features from source code. It is designed as an iterative and rule-based process.

- The behavioral abstraction tool extracts a specification of the dynamic behavior of the components identified during the structure abstraction process. It also works from static analysis of the source code.

The general approach envisioned in the Econet project is that of a tool box used in an iterative and interactive process. Each iteration applies one of the abstraction tools (the two described above or others) to the (possibly annotated) source code and produce the same source code with new annotations. The idea is to combine primitive transformations in a customized human driven process. Examples of primitive transformations are: annotate a Java program from user information, build a component model from a plain or an annotated Java source, analyze a distributed program to detect components (deployment), analyze dependencies using graph tools and extract clusters, extract behaviors from communications between components, ...

[1] Nr 16293RG, www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:materials

As already stated, this paper focuses on one basic tool of the framework: recovering components and communications from a plain Java application. The input is a (possibly annotated) Java source code, and the output is a set of components with several kind of relations between them (communications, links, inheritance) and a set of data types.

3 Component and Communication Abstraction from Source Code

A full component based architectural model can include: Components, required and provided services of the components, composition structure of the components (composite components and sub-components), communication channels between components, and data circulating on the communication channels. All these informations are not independent, the composition structure of the components limits the possible communication channels, data circulating depend on the needs of the services, ... We will discuss some heuristics to extract most of these informations in section 3.2. Before that, we set some hypotheses we made on the analyzed code.

3.1 Working hypothesis

We are currently working on Java code, but our hypotheses could apply to any Object-Oriented language. Component types are created from the classes and Java interfaces of the system, we do not group variables, routines or object to create components as traditional component recovery would do (*e.g.* see [**?**]). Classes and Java interfaces are called *types of interest*. A type of interest may be either a component type or a data type.

We pay more attention to the components because they are semantically richer from an architectural point of view: they have provided and required interfaces, they communicate, etc. We made the decision to favor precision of components over recall, this means, we prefer recover fewer components, with lower probability of false positive, rather than trying to extract "all" components at the cost of having many false positive. Therefore, we use strict rules for components. The idea is that the iterative nature of the process (see Section 2) may allow to extract more components.

In this paper, we work with systems that were designed with componentization in mind (see Section 4.1). A "random" application may not offer the notion of a component (as defined in Section 2.1) without heavy restructuring. However, note that other used the same rule (see [**?**]), and that even in this case, our tool present interest as explained in Section 2.2.

3.2 Heuristics for recovering component types and communications

Components:

1-a) If a type of interest is passed as parameter of a method or returned by a method it is considered a data type, otherwise it is considered a component type.

The idea is that static checking of communication integrity is possible only when all uses of a component are explicit (as opposed to accessing the component through a pointer). This is a standard assumption in CBSE (*e.g.* [**?**], [**?**]).

1-b) Exception to heuristic 1-a): component may be passed to or returned by constructor.

This is often necessary to allow building composite components.

1-c) A sub-type of a data type is considered a data type.

This follows from heuristic 1-a), since instances of the sub-type could be used as parameters using sub-typing rules.

1-d) "External" types of interest (not defined in the Java project) are ignored.

We chose to ignore all types of interest not defined in the Java project (*e.g.* ignore all external libraries as `java.io.*`, or `org.eclipse.*`). One reason is that we want to extract the provided services of the components, and their structure. This requires having access to the source code (the Java reflective API could help, but we favor a more generic solution). Also, there are good chances that `Object` will be passed as parameter of, or returned by, some method, turning it into a Data Type (rule 1-a)). This would, in turn, qualify all types of interest as Data Types (rule 1-c)).

Finally we cannot hope to restructure the entire world and would like to limit ourselves to the application at hand.

Composition structure:

2-a) The composition structure of components is extracted from the fields (component that is part of another component).

We choose to consider the maximal structure, that is collecting the defined attributes and the inherited ones.

Communications:

3-a) There is a communication between two component types if a method of one component type makes a call to a method of the other.

Other communications means are possible. For example in Java, using the reflective API. We did not consider these cases here as they require more advanced knowledge of an application to know how communications are implemented in it.

If the method "returns" `void` it is a one way communication, otherwise, it is a two ways if it returns a data type.

Subtyping:

4-a) Subtyping relationships are computed from the language inheritance relationships.

In Java, there are two such relationships: `extends` and `implements`. Component types may inherit from component types but not data types (see also rule 1-c)). Data type may inherit from data types or component types.

Note that CBSE usually makes little use of inheritance. We decided to deal with it because some implementation may use it to represent component types and their communication interface, for example.

Required/provided services:

5-a) The required services of a component type are those methods that are called in the component type.

5-b) Provided services are all the publicly available methods defined in the component type.

In Java, these methods are the `public` and `default package` ones.

3.3 Further considerations

With these rules it is also possible to do some checking of the proper communication channels established. The *boundary analysis* checking has the role of identifying the communications which are not conform to the structure of the components. In a strict component framework, components may communicate directly if they are in the same scope or boundary of a composite structure. A component may communicate with its composite component or with sibling components (see Section 2.1). But the Java code may not always respect these well-formedness constraints. By identifying communication channels and structural relationships between component types, we are able to pinpoint such anomalies. The correction of the problem would be left to the user in most cases although some could be performed automatically, when the communicating components have an ancestor in common that is not their direct parent.

Java has four notions of nested classes, *static member*, *inner*, *local* and *anonymous*. Most of the time these nested

classes are used to implement complex things, like simulating multiple inheritance. However it is theoretically possible to define some public services using this feature. It may, therefore, be important to analyze them to get the whole story. Our plugin is able to analyze nested classes as normal classes. However because we are not sure how exactly they could be used in a componentized implementation, we suggest to ignore them for now until more information is gathered on the possible uses. There were no inner classes in the systems we experimented with.

4 Experimentations

The above rules apply under some hypotheses on the Java code.

We use the Eclipse JDT parser to analyze Java projects. The program is a source code (binary Java code could be analyzed in more or less the same way, either directly or after decompiling). We don't consider generic types, this is a main future extension of this work. To relax this restriction is not easy since Java 1.5 introduces a sophisticated type system and we have to distinguish generic definition, instantiation of them, their use in fields, inheritance and methods.

Figure 1 illustrates the output produced by our plug-in. The graph may be dynamically configured to show component types and/or data types, structure relationships, and/or communication relationships, and/or inheritance relationships. Here the graph shows the component types and the data types, and the structure relationships and communication relationships.

4.1 The test bed

We experimented our tool on various implementations of CoCoME. CoCoME[2] [**?**], the Common Component Modeling Example, is a contest set to evaluate and compare the practical appliance of existing component models and the corresponding specification techniques using a common component-based system as modeling example. Based on a UML-based description of CoCoME, a provided sample implementation, and test cases, the participating teams had to elaborate their own modeling of CoCoME, applying their own component model and description techniques. The example describes a Trading System as it can be observed in a supermarket handling sales. This includes the processes at a single Cash Desk like scanning products using a Bar Code Scanner or paying by credit card or cash as well as administrative tasks like ordering of running out products or generating reports. CoCoME specifications include a use case model, detailed specification of use cases (both in textual form and as sequence diagrams), architectural component models, a deployment model, or test cases.

[2]available at `http://www.cocome.org/`

We used three different instances of CoCoME. Each instance is composed of a meta-model and its implementation. Except for the first instance, the reference sample implementation, each instance uses the specificities of the tool and approach championed by a given team. For lack of time, many team did not implement the entire solution.

The implementations we used for our experiments are:

Reference: This was the basic sample implementation provided with the system specifications as a reference. It does not make use of any specific component technology and consist in a Java program designed and implemented manually (no automatic generation). It includes 5078 LOC, 40 packages, 95 classes, 20 interfaces and 375 methods.

The abstract model is that of CoCoME. It is relatively detailed with components, sequence, or communication diagrams. However it does not explicitly identify all the services, and only a few appear in sequence diagrams.

rCOS: A solution implemented at the United Nation University, Macao (`http://wiki.iist.unu.edu/cocome`). It includes 2772 LOC, 28 packages, 58 classes, 0 interfaces and 291 methods.

Oasis: The solution proposed by the Oasis research team from INRIA Sophia Antipolis, France (`http://www-sop.inria.fr/oasis/Vercors/Cocome/`). As described in [**?**], this approach is based on a component model for distributed components called *GCM for Grid Component Model*, that is an extension of the FRACTAL Component Model to address Grid concerns. The implementation is done with the Java middle-ware ProActive, an extension of Java with active objects communicating `via` asynchronous calls and futures. Code for the composite is automatically generated from the architectural description. The approach allow the specification of the dynamic behavior and from this description the control code is automatically generated. The user has to provide the code for the services.

It includes 1770 LOC, 13 packages, 26 classes, 26 interfaces and 194 methods.

These three implementations were chosen for the following reasons: The first one is the reference implementation and it seems natural to consider it. The two others made their source code and abstract model available on Internet which was a requisite for our analysis.

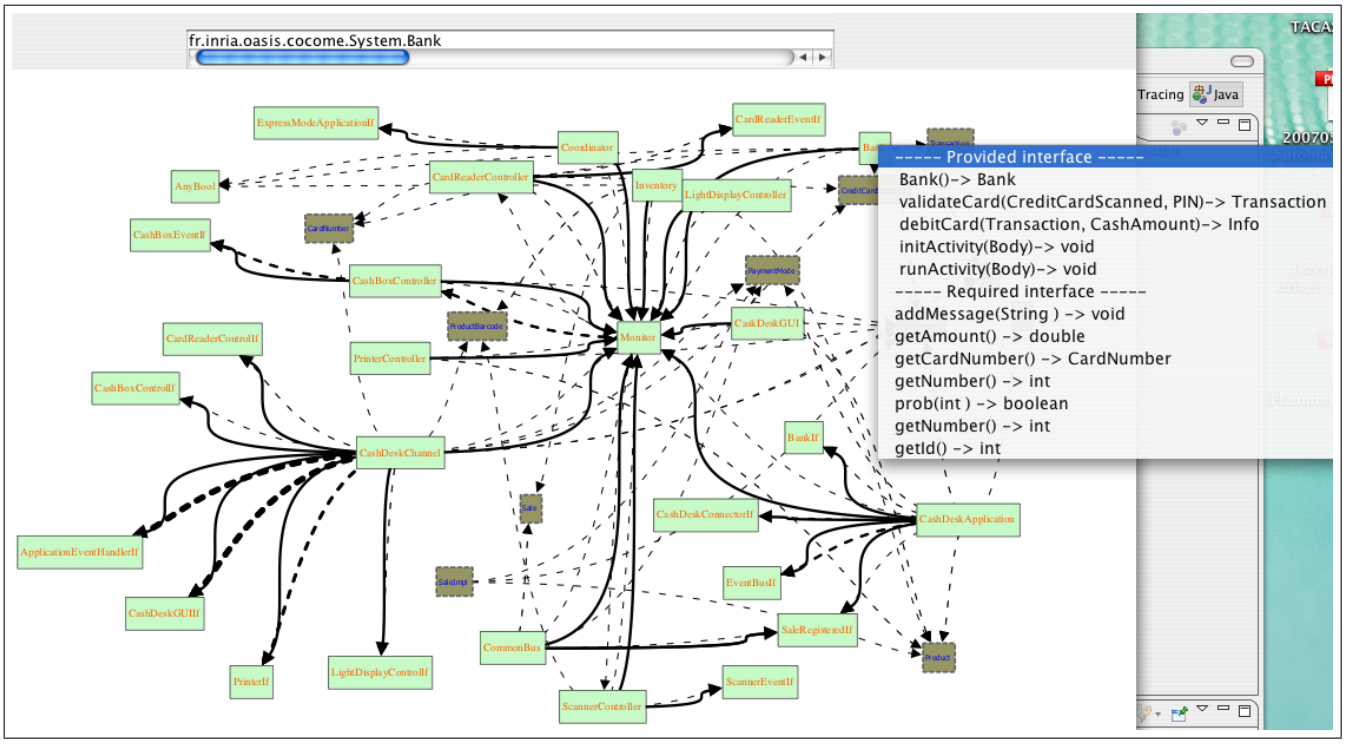

Figure 1. A screenshot of the result presented by our tool. The dark box are data types, the light boxes are component types. Plain arrows go from a composite component type to a sub-component type. Dashed arrows illustrate a communication (*i.e.* one or more method call), from the caller to the callee. The width of the dashed arrow depend on the number of services called on this communication channel. A context window (on top of the main window on the right hand side) on the components show the services it provides and requires. A context window on the communication shows what services are involved in this communication.

4.2 Results

We will provide and comment some results on our three experiments in the following. Because numbers are relatively few and small, we chose to present raw results instead of statistics such as precision and recall. An other issue to keep in mind is that comparisons are difficult since in different cases, the implemented model (the code) does not conform exactly to the abstract model (the documentation), *i.e.* the abstract model was not implemented as specified. In such a case, the comparison lead to bad results, not because of our tool, but because of the gap between specification and implementation. It is one of the goal of our research to detect such a gap.

4.2.1 CoCoME Reference

There are 25 components in the abstract model and our tool extracted 51. After analysis, we found that from these 51, 12 are components in the abstract model, 9 correspond to communication interfaces (with 9 implementations), 5 are false positives (3 data events and 2 exceptions), and the 16 others are actual components types not mentioned in the abstract model. Note that the 2 Java exceptions could easily be identified properly by the tool because they inherit from the Java class `Exception`. This could be a future improvement. Identification of components was made difficult by the fact that they are implemented with packages. This made it impossible in general for our tool to recover them since we don't analyze packages. However, some of these packages contain only one Java class that implements the component. These are the 12 component types from the abstract model that we could recover.

Nine other component types could have been detected (they have a corresponding single class) but were not because of the use of a Factory design pattern. The Factory is used to create the components and therefore it has methods returning these components. This goes against rule 1-a) and

the tool classified them as data types. We discuss this issue in Section 4.3.

For the services, it is impossible to know how many there are in the abstract model because they are not all identified. The tool found 481 provided services and 321 required services. They correspond respectively to all public methods and all called methods in the application. This is clearly a problem and we are looking into it (Section 4.3).

Finally 139 communication channels (one-way) lead to less than 70 links (two-way) while there were 63 manually identified communication channels in the UML diagrams. We feel this result is quite acceptable.

On a more qualitative analysis, this is one example where the mapping between the abstract and concrete models was made difficult by some design decision: to implement the components as Java packages. For example, the special component `EventBus` of the abstract model is replaced by the *Java Message Service* framework, and the database uses *JDBC*.

On a positive note, we could detect some design patterns that were used but not documented. For example, in the control part (`CashDeskLine`) the developers often applied a design pattern to implement the controller components of the `CashDeskLine`. While in the data management part (`Inventory`), others patterns are used.

4.2.2 CoCoME OASIS

The abstract model does not cover the complete case study. The article [**?**] only describes in details the architecture of the `CashDesk`.

There are 9 components in the abstract model, 1 large composite component type (`CashDesk`) and 8 subcomponents. There are 60 services identified (60 provided and 60 required, since, in the abstract model, a service required by one component type must be provided by another). Our tool found 7 of the nine components although they did not always had exactly the same name as in the abstract model. The two missing are the large composite component type which is not implemented explicitly in the code, and another component type (`logger`) that was not implemented either. This last one may have been substituted by a `Monitor` component type (note: `Monitor` is the components type part-of 12 other components in the center of Figure 1).

The plug-in extracted 21 component types (from which 7 were in the abstract model). The others are related to implementing a running application with GUI, DataBase access, etc. (*e.g.* `Main`, or `CommonBus`). There are also components implementing parts of the CoCoME reference model, but not shown in the OASIS abstract model (*e.g.* `Inventory`). Finally there are also a number of data type that seem to be additional component types not listed in the abstract model (data type related to transaction, pin code, and implementations of components).

We feel there is little to do with the first issue, running application will presumably always require such utilitarians and it seems a waste of time to model this in a component model. For the second issue, we see it as a clue from our plug-in that the implemented code does not match completely the abstract model. This was the kind of information we were looking for.

Because the large composite component type was not implemented, we had no success in recovering the structure relationships of the abstract model.

There are 60 services in the abstract model, but only names are given (no parameters and return type), 53 of these services were found by the plug-in with complete signature. Amongst the 13 communication interfaces we found 9. The remaining 4 interfaces were found but are empty (no services in them).

On a more qualitative analysis, the plug-in extracted 13 types of interest implementing interfaces (coherent group of services) used for communication. Their names are suffixed by `If`. When a component type provides or requires an interface, its name is suffixed by `ControlIf` and `EventIf` respectively. The (component) interface itself is implemented as a Java interface.

We could also identify an implementation following the membrane concept inherited from Fractal [**?**].

Although we were able to recover most of the services listed in the abstract model (53 out of 60), we found that some of them were relocated. For instance, `BankIf` has two services `validateCardReq` and `validateCardResp` but the implementation has only one `validateCard(CreditCardScanned, Pin) -> Transaction`. This service implements both of the services listed in the abstract model.

The tool extracted 194 provided services and 111 required services. This is due to the large amount of public methods that are needed in Java to implement a real application. This is a problem and we are studying another solution (see Section 4.3).

4.2.3 CoCoME rCOS

We first compare the rCOS implementation with its documented UML2 component diagram. It specifies the interfaces and services between components, as well as some Java classes inside the component types. There are 11 components in this diagram from which our tool extracts the only 4 components that were actually implemented in the code. There are 25 services in the component diagram and 22 of them are implemented in the source code. All these services were found, but there were also a lot of false positives due to the following reasons:

- too many methods are public in the Java code, such as methods for initializing GUI components or managing Mouse Events,

- the classes (*e.g.* `CashDesk`) that have been "promoted" to component status have methods that would not have been part of a proper component model.

Again, these are clearly problems with the implementation.

It became rapidly clear that the code did not implement the abstract model faithfully. To try to get better result, we resorted to a second diagram ("*components in the use cases*"), that describes some additional components, without specifying services, or communication channels. This second diagram has 16 components (2 of them in common with the first diagram). Twelve of these were implemented in the code, and the tool found 10 of them. Eight component had the same name as in the abstract model and two had very different names (*e.g.* `StoreGUI` plays the role of the `Application` subcomponent). Another component have been implemented differently: 6 components actually simulate the `database` component. It looks like the second diagram has been influential in the structuring of the application. The only component implemented that was missed was the `Store`.

In total, from the 58 Java classes in the code, our tool found 29 components and missed only one, 18 components types were found in the abstract model (with its two diagrams). As described in [**?**] many components have been implemented using an abstract class and a concrete subclass suffixed by `Impl`. However, no java interface have been used to represent the interfaces of the components. Our tool currently detects both the abstract class and the concrete class as components, generating 18 components which should be merged as 9.

All the communication channels implemented have been discovered except the ones of the `Store` component which was not detected as such. Some channels have been merged as a consequence of the absence of components in the implementation. For instance, in the specification, the `CashDeskGUI` communicates with the `SalesHandler` using its `CashDeskIf` interface and through the `BusControler` component which uses the `SalesIf` interface to forward messages to the `SalesHandler`. The absence of implementation of the `BusControler` has the consequence of merging the `SalesIf` and `CashDeskIf` and their channels.

Results are therefore bad if one only considers the UML Component diagram, but this mainly reflects the fact that the implementation was not structured according the abstract model described in this diagram.

4.3 Discussion and Future Work

Our first conclusion is that the results are quite encouraging. Although the rules to recognize component types may seem very strict, in the chosen context (application developed with componentization in mind), they worked quite well. We could very quickly discover mappings between the concrete code and the abstract model which was one of our goals. The tool also highlighted big mismatches between the designed application (abstract model) and the implemented one. More specifically in the case of the rCOS implementation.

One difficulty was raised by the use of Factory and methods to look for specific components. Although these uses are legitimate, they result in methods returning component types. With our rules (especially rule 1-a)) such components were classified as data types. This is also a consequence of us working specifically on component uses in this first experiment. We need now to pay more attention to components construction.

We see two solutions to this problem, both relying on the iterative and interactive nature of the extraction process. The tool can build its result on top of previous iteration through the use of annotations in the source code. These annotations (inserted by the tool or some other one from our tool box, see Section 2.2) currently flag a type of interest as a component type or a data type:

- The simplest solution would be to implement a manual editor to flag a type of interest as the user sees best fitted. In the case of the Factory, a data type would be re-flagged as a component type. The tool would then need to rerun to make the needed adjustment (for example because of the data type inheritance rule 1-c)).

- A more strict solution, could be to create a "`@IgnoreReturn`" annotation that would work somehow as the "`@SuppressWarnings`" annotation in Eclipse. It would cause the tool to ignore a return type and thus allow to correctly flag the type of interest as a component type. This solution is more cumbersome for the user because it imposes to find all the methods legitimately returning a component type. On the other hand, it would be more semantically correct.

For now we tend to prefer the simplest solution although it is not implemented yet.

Some implementation elected to implement component types as Java packages. This is a problem for us as we do not analyze packages yet. The fact that in rCOS, the tool was able to find the main class out of the many that implemented a component type, might point toward a possible solution. We are looking into this problem, but no immediate solution came to mind.

We have had rather less success with the communication and required/provided services. This is due to the many public methods that are required in a Java application to have all classes communicate between themselves. A possible solution that we are studying is to restrict the provided and required services only to the identified components. Currently, we consider all classes because components may need to access methods in the data types. A first experiment on OASIS gave only 75 required services instead of the 194 currently extracted. This seems therefore a path worth pursuing.

5 Related Work

In CBSE reverse-engineering, the concepts of component and architecture vary from one approach to another. For example, in [**?**], design components are high level concepts close to design patterns, but they are not abstract components.

Chouambe *et al.* have objectives very similar to ours, however, they chose to extract the components from metrics instead of using heuristics. They put much more emphasis than we do in discovering a "proper" components' structure. It is too early to provide a more significant comparison of these two approaches.

Washizaki *et al.* [**?**] propose to extract JavaBeans components of Java programs. Only the structure is abstracted, while, in addition, we consider the communication integrity property. Our analysis of the composite structure is comparable to their structural clustering algorithm but we simplify by assuming that component structure is built from the class fields.

There as been a lot of research on architecture and component recovery in the reverse engineering community (see Koschke, [**?**] and [**?**], for a review of the field). However, the problems they tackle are different:

Architecture recovery typically tries to partition the set of software elements of a system in various coherent subsets. For this, it may use clustering to group together the elements that have more things in common.

Component recovery typically tries to group constants, variables and routines of a procedural source code into objects or classes. For this it considers what variables or constant routines access, and may also use clustering of variables and constants.

This is far from our preoccupations, as we work from an object-oriented system assuming that the classes correspond (or may correspond) to components. We are not trying to regroup things together, but rather to explicitly identify communication paths between existing components.

Favre *et al.* [**?**] describe how they (manually) build a meta-model from a component based source to help understand the system and help build reverse engineering tools for that system.

In [**?**], Abi-Antoun *et al.* manually re-engineered a Java legacy system into an ArchJava system with explicit (and enforceable) definition of control flow and data sharing. This correspond closely to our final objective although the work presented here is only the first step toward this objective. There are two main differences between our work and their: they did a manual re-engineering whereas we explored automatic tools; and, the work from a conventional object-oriented system, whereas we are assuming an implementation that is already "component aware" in the sense that we suppose the system was developed from some abstract component definition although the implementation may not perfectly match this abstract definition.

Component recovery and *architecture recovery* are the main issues for abstracting structures. In their paper Bowman *et al.* [**?**] study various ways to extract model information from Java code. Two approaches are possible: static or dynamic analysis. Static analysis usually gives more abstract information. Dynamic analysis depends on the execution context and may provide very accurate information about polymorphic call, dynamic types of objects, and information related to the use of the reflective Java API. However, it usually produces a huge quantity of information that one must filter. Static analysis also allows exploiting comments and code annotations which we are using in our tool.

The model looked for by [**?**] is a simple entity relationship model but it is not too far from a component model. The conclusion from [**?**] is that: if static analysis is sufficient thus disassembling (Java Byte code) is probably the best choice. The problem is still open. For instance, [**?**] considers that runtime analysis or profiling is needed, since types and objects may be dynamically created.

6 Conclusion

One possibility to fight against the erosion of a system's architecture is to make it explicit in the source code. Component Based Software Engineering proposes tools and approaches that allow this. They specify explicitly which component may communicate with which other, and they offer the possibility to check this property either statically or at runtime. However, existing approaches remain mostly theoretical, and industrial approaches such as CCM, EJB, or OSGI focus on a strong and mature runtime infrastructure, but provide little support for automatic verification of the correctness of components' usage. It may often happen that even if it is specified with a rich component model, an application is implemented as a flat component structure that do not allow checking the integrity of communication

for example. There is a mismatch between the component specification and the component implementation.

The Econet project aims at defining a recovery process and tool box to help mapping the implemented component code to the specified abstract model. One of the tools of this toolbox is a component recovery tool that extracts component types, data types, provided and required services, structure of composite component types, and communication channels between components.

This tool is intended to help its user compare (and map) a concrete implementation with an abstract model. We saw that one of our application (CoCoME rCOS implementation) had a bad mapping in this sense. It could also be used to check the good state of the architecture of a system by indicating when one components is used improperly (*e.g.* passed as parameter, or communicating with the wrong other component), something that is typically not possible with existing industrial approaches.

The tool may also be used to hint at possible problems in the implementation: component passed as parameters, cycle in the structure of (composite) components, boundary analysis checking (see Section 3.3), etc.

Finally we believe the tool could be used to help restructuring an application into a componentized one. It could help the user identify components or check that what he thinks are components really respect the typical rules (see Section 3.2) of the kind.